	Type	L #	Hits	Search Text	DBs	Time Stamp	Comments
1	BRS	L1	0	exception same decompressed adj instruction same address	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/30 10:47	
2	BRS	L2	1	exception same (decompressed adj instruction)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/30 10:49	
3	BRS	L3	0	exception same (decompressing adj instruction)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/30 11:14	
4	BRS	L4	0	exception same (decompressing adj data)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/30 10:50	
5	BRS	L5	44	exception same decompress	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/30 10:51	
6	BRS	L6	481770	5 and address and compressed instruction	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/30 10:52	
7	BRS	L7	0	5 and address and compressed adj instruction	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/30 10:52	

	Type	L #	Hits	Search Text	DBs	Time Stamp	Comments
8	BRS	L8	0	5 and instruction adj address	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/30 10:53	
9	BRS	L9	27	5 and address	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/30 10:53	
10	BRS	L10	264	6 and (exception adj handling adj routine)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/30 10:54	
11	BRS	L11	0	9 and (exception adj handling adj routine)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/30 10:55	
12	BRS	L12	2	decompress and (exception adj handling adj routine)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/30 10:59	
13	BRS	L13	5	non-native adj instruction and (exception adj handling adj routine)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/30 10:58	
14	BRS	L14	1	non-native adj instruction same (exception adj handling adj routine)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/30 11:00	

	Type	L #	Hits	Search Text	DBs	Time Stamp	Comments
15	BRS	L15	0	decompress same (exception adj handling adj routine)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/30 10:59	
16	BRS	L16	1	decompressed same (exception adj handling adj routine)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/30 10:59	
17	BRS	L17	5	non-native adj instruction and (exception adj handling adj routine)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/30 11:00	
18	BRS	L18	0	17 not 13	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/30 11:01	
19	BRS	L19	4	17 not 14	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/30 11:01	
20	BRS	L20	0	exception same misalligned adj instuction	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/30 11:15	
21	BRS	L21	0	exception same misalligned adj address	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/30 11:15	

	Type	L #	Hits	Search Text	DBs	Time Stamp	Comments
22	BRS	L22	2	exception same misaligned adj address	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/30 11:26	
23	BRS	L23	17	exception same ((misaligned or improper) adj address)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/05/30 11:31	

Grow & Fold: Compression of Tetrahedral Meshes

Andrzej Szymczak and Jarek Rossignac

Graphics, Visualization & Usability Center
College of Computing, Georgia Institute of Technology
Atlanta, GA, 30332-0280, USA

e-mail: andrzej@cc.gatech.edu, jarek@gvu.gatech.edu

1 Abstract

Standard representations of irregular finite element meshes combine vertex data (sample coordinates and node values) and connectivity (tetrahedron-vertex incidence). Connectivity specifies how the samples should be interpolated. It may be encoded as four vertex-references for each tetrahedron, which requires $128m$ bits where m is the number of tetrahedra in the mesh. Our 'Grow&Fold' format reduces the connectivity storage down to 7 bits per tetrahedron: 3 of these are used to encode the presence of children in a tetrahedron spanning tree; the other 4 constrain sequences of 'folding' operations, so that they produce the connectivity graph of the original mesh. Additional bits must be used for each handle in the mesh and for each topological 'lock' in the tree. However, as our experiments with a prototype implementation show, the increase of the storage cost due to this extra information is typically no more than 1-2%. By storing vertex data in an order defined by the tree, we avoid the need to store tetrahedron-vertex references and facilitate variable length coding techniques for the vertex data. We provide the details of simple, loss-less compression and decompression algorithms.

2 Problem Statement

This paper addresses the problem of a bit-efficient lossless encoding of the incidence of a tetrahedral mesh whose boundary is a manifold surface. A simple representation of such a mesh consists of two tables: the *vertex table* keeping vertex coordinates and vertex data, such as temperature or pressure, and the *tetrahedron table* storing quadruples of vertex indices, representing vertex sets for each one of the m tetrahedra in the mesh. The tetrahedron table describes explicitly only the vertex incidence for each tetrahedron. However, all other connectivity information, like tetrahedron-face or triangle-vertex incidence can be derived from it algorithmically. For a mesh with one Million vertices and six Million tetrahedra, the tetrahedron table requires $128m \approx 7.68 \times 10^8$ bits if 4-byte pointers are used to reference vertices or $80m \approx 4.8 \times 10^8$ bits if the vertex references are stored as 20-bit integers crossing the byte boundaries. The total size of the vertex coordinates and data (12-bit coordinates and 16-bit for a single scalar value) of such a mesh amounts to 5.2×10^7 bits: almost 10 times less. Therefore, the connectivity information dominates the storage cost and it is important that it is compressed. In this paper we are concerned only with the compression of this incidence information (described by the tetrahedron table); we do not discuss compression of the vertex data.

Our coding algorithm takes a tetrahedron table as input and produces its encoding - a string of about 7 bits per tetrahedron, thus achieving a 10-to-1 compression ratio for common meshes. The decoding algorithm is able to produce a tetrahedron table based on that string. The decoded mesh will be identical to the original one, but its tetrahedra and vertices will be listed in a different order. Even without applying any compression scheme to the vertex coordinates and data, our algorithm is able to encode both connectivity and geometry of our one Million vertex mesh using about 9.4×10^7 bits, achieving the compression ratio of about 5.7/1. Furthermore, because our scheme permits to transmit and decode the incidence independently of the vertex information, it makes it possible to use various prediction-based techniques to compress the vertex location and data ([25], [11], [12], [4]).

Fifth Symposium on Solid Modeling Ann Arbor MI

3 Prior Art

3.1 Mesh Representations

Numerous data structures have been proposed that combine adjacency and incidence information (see [20] for a review). Examples include winged-edge representation ([1],[2]), face-adjacency hypergraph ([7]), half-edge structure ([16],[13]), radial-edge structure ([8],[9]) and selective geometric complex ([19]). The goal behind the design of these data structures is to provide an efficient way of accessing different kinds of adjacency information without taking up much storage space. A common idea is to store some of the adjacency relations (possibly in a partial form) explicitly and use them to derive the ones which are not explicitly stored. For example, in the winged-edge representation, each face and vertex point to one of the adjacent edges and each edge – to its endpoints, the two adjacent faces and the four edges adjacent to that edge and the neighboring two faces. Using the above relations it is possible to compute all others (for example, all edges adjacent to a face or all edges adjacent to a vertex) in time proportional to the *local* complexity of the mesh (for our two examples, the number of edges of the face and the number of edges out of a vertex, respectively). One of the concerns of the boundary data structures is to reduce storage space needed to keep adjacency information. However, they take up a lot of space since they also attempt to minimize the time needed to access adjacency information. Therefore, it is not fair to treat boundary data structures as compression schemes. In fact, as shown in [29], such a data structure requires at least $4E$ pointers, where E is the number of edges of the mesh – more than a tetrahedron table.

3.2 Mesh Compression Schemes

Staadt and Gross [23] and Trotts et al. [27] independently propose a tetrahedral mesh simplification process, which removes tetrahedra by collapsing their edges in a sequence that attempts to minimize, at each stage, the error computed using different cost functions. Such a simplification may be viewed as a lossy compression technique and complements our loss-less compression, which may be used to compactly encode the simplified meshes.

We are not aware of any other work in compressing tetrahedral meshes. However, several approaches that have been proposed for compressing triangle meshes in 2D or 3D could inspire new approaches for compressing tetrahedra.

Deering's approach [4] is a compromise between a standard triangle strip and a general scheme for referencing any previously decoded vertex. Deering uses a 16 register cache to store temporarily 16 of the previously decoded vertices for subsequent uses. One could envision extending the notion of a triangle strip to tetrahedra. Keeping 3 registers for the last 3 vertices used, each new tetrahedron will be defined by these 3 vertices and a fourth vertex either new (the next vertex received in the compressed input stream) or previously received (and identified by its location or id in main memory or cache). One of the vertices in the registers will be replaced by the fourth one and the operation repeated. Unfortunately, we do not know of simple and efficient algorithms for identifying the suitable sequence of tetrahedra.

Hoppe's Progressive Meshes [11] permit to transfer a 3D mesh progressively, starting from a coarse mesh and then inserting new vertices one by one. Instead of a vertex insertion to split a single triangle, as suggested in [6] for convex polyhedra, Hoppe applies a vertex insertion that is the inverse of the edge collapse operation popular in mesh simplification techniques [12], [18], [10]. A vertex insertion identifies a vertex v and two of its incident edges. It cuts the mesh open at these edges and fills the hole with two triangles. The vertex v is thus split into two vertices. Each vertex is transferred only once in the Hoppe's scheme.

Hoppe suggests that it may be possible to extend this scheme to tetrahedra. Each vertex split would require identifying one vertex of the current (simplified) version of the mesh and a cone of incident triangles. As the vertex is extruded into an edge, these triangles would be extruded into new tetrahedra. The cost of this approach is the identification of each vertex ($\log_2 v$ per vertex) and the identification of the cone of incident edges, which on average would require 15 bits per vertex. Assuming that the ratio of the number of tetrahedra to the number of vertices is about 20/3, this approach would require roughly $(3\log_2|V| + 45)/20$ bits per tetrahedron.

The Topological Surgery method developed by Taubin and Rossignac [25] builds a vertex spanning tree that splits the surface of the mesh into a binary tree of corridors (generalized triangle strips). The two trees are encoded using a run length code, which for highly complex meshes yields an average of less than two bits per triangle. In addition, one bit per triangle is used to indicate whether the next triangle in the corridor is attached to the left or to the right edge of the previous one. For pathological cases, with a non-negligible proportion of multi-child nodes in the two trees, the above approach no longer guarantees linear storage cost. The application of this technique for VRML files is discussed in [26]. Taubin and Rossignac's technique could be extended to tetrahedral meshes by encoding the tetrahedron spanning tree (as we do) and then by encoding the boundary and the 'cut' which is a two dimensional non-manifold triangulated surface. In some sense, our scheme offers a compact encoding of this surface.

Inspired by [15] and improving on [17],[22], Denny and Sohler proposed a technique for compressing *planar* triangulations of sufficiently large size as a permutation of its vertices [5]. They show that there are less than $2^{8.2|V|+O(\log_2|V|)}$ valid triangulations with $|V|$ vertices, and that for sufficiently large $|V|$, each triangulation may be associated with a different permutation of these vertices. Their approach requires transmitting an auxiliary triangle that contains all the vertices and the vertices themselves in a suitable order computed by the compression algorithm. Although for sufficiently complex models the cost of storing the mesh incidence is null, the unstructured order in which the vertices are received and the absence of the incidence graph during their decompression makes it difficult to use predictive techniques for vertex data encoding. We believe that this approach may be directly adapted to tetrahedral meshes. However, as in the 2D case, it will make it difficult to compress the vertex data, because the connectivity of each new vertex is derived from its position and hence cannot be used to estimate the position.

Edgebreaker, introduced by Rossignac [21], allows to compress the connectivity of a triangular mesh using only about 2 bits per triangle. Similarly to Grow&Fold, the compression starts with a depth-first search traversal of the dual graph of the mesh. The traversal is topological, i.e. after a triangle is discovered we first visit the triangle adjacent along its right edge whenever it is possible (in the Grow&Fold scheme, the traversal order is arbitrary). Whenever a new triangle is discovered, it is classified as one of five possible types according to which of its edges are shared with triangles which remain undiscovered. A variable length encoding technique is then applied to encode the sequence of types of triangles encountered during the traversal using about 2 bits per triangle. That sequence of triangle types turns out to be sufficient to reconstruct the original mesh. In principle, the Edgebreaker could be extended to 3D case. However, it will no longer be that simple. For example, sometimes extra information would be needed to encode the offset of the fourth vertex when a new tetrahedron without any new vertices is added to the mesh during decompression. Also, the number of ways in which a removal of a tetrahedron can split the mesh into connected components is considerably larger than in the 2D case. Grow&Fold seems to be a simpler and cleaner alternative.

Other compression schemes for planar graphs and triangulations are discussed in [28] and [14].

4 Compressed Format

Our encoding of a tetrahedral mesh consists of two parts:

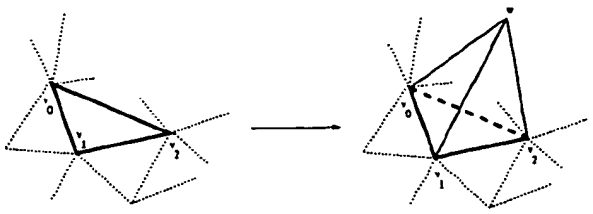

Figure 1: *The attaching operation. $v_0v_1v_2$ is an external triangle of the starting mesh and w is a new vertex.*

- The *tetrahedron spanning tree string*, defining a tetrahedron tree – a complex containing all tetrahedra appearing in the encoded mesh and some of the incidence relations.
- The *folding string*, defining how to uncover incidence relations absent from the tetrahedron tree by means of folding and gluing operations.

4.1 The Tetrahedron Spanning Tree String

A tetrahedron tree is a three-dimensional simplicial complex which can be obtained from a single tetrahedron as a result of *growing*, i.e. incrementally applying the operation of attaching a tetrahedron to an external face. As shown in Figure 1, attaching a tetrahedron to an external face with vertices v_0, v_1 and v_2 is equivalent to creating a new vertex w and adding a tetrahedron with vertices v_0, v_1, v_2 and w to the mesh. The tetrahedron tree string stores information about which external faces are *attachable*, i.e. to which of them tetrahedra are attached later in the growing process. Right after each attaching operation, the decompression procedure reads a triple of bits of the encoding string and marks each of the three new external faces (v_0v_1w, v_1v_2w and v_0v_2w on Figure 1) as either attachable or not, according to the value of the corresponding bit of the triple. The tetrahedron tree string consists of one triple of bits per tetrahedron, which yields a total of $3m$ bits, where m is the number of tetrahedra in the mesh.

4.2 Folding String

Using the tetrahedron spanning tree string, the decoding algorithm is able to grow a tetrahedron tree with a tetrahedron table $\mathcal{T}'$, having m rows and referencing $m+3$ vertices where m is the number of tetrahedra in the original mesh $\mathcal{M}$. The tetrahedron tree can be thought of as the result of cutting $\mathcal{M}$ along the surface formed by *cut triangles* (defined in the next section). Therefore, tetrahedra of the tree correspond to tetrahedra of $\mathcal{M}$ in a one-to-one fashion and each external triangle of the tree either corresponds to an external triangle of $\mathcal{M}$ or

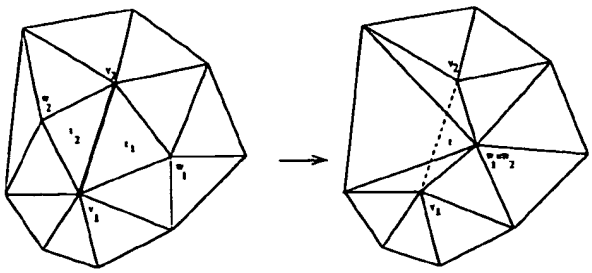

Figure 2: *The folding operation as seen from the outside of the mesh. v_1v_2 is the fold edge of both t_1 and t_2 and w_1 and w_2 are the equated vertices. After the equating is done, t_1 and t_2 have the same vertices and therefore become one internal triangle t.*

belongs to a pair of triangles corresponding to a single cut triangle of $\mathcal{M}$. In order to reconstruct the mesh $\mathcal{M}$ from the tree we must identify the triangles belonging to the same pair. We do it by incrementally applying *gluing* and *folding* operations. A folding operation (Figure 2) 'folds' the boundary of a mesh at an edge. It can be executed only if that edge is the *fold edge* in both external triangles adjacent upon it. As a result, the two incident triangles are identified and become an internal face of the mesh and their two vertices (the ones that bound the two incident triangles but not their common edge) are equated. The folding operation changes the adjacency of nearby faces, so it may make two triangles of the starting mesh which do not share an edge be adjacent along a fold edge. Such triangles are identified by a fold operation later during decompression. Thus, the way in which fold edges are assigned to external triangles (later referred to as the folding scheme) imposes restrictions on the order of execution of folding operations. It determines the way in which the vertices are equated uniquely. However, there may be several sequences of folding operations consistent with it and hence leading to that particular way of equating vertices (see Figure 3).

The need for the gluing operation, which identifies two arbitrary external triangles, arises when two external triangles do correspond to the same triangle of the mesh $\mathcal{M}$ but never become adjacent along the fold edge in both. Being more general than the folding operation, gluing operation alone suffices to construct $\mathcal{M}$ from the tree. However, the advantage of the folding operation is that it is cheaper to encode.

The folding string associates two bits of information with each external triangle of the tetrahedron tree given by the tetrahedron table $\mathcal{T}'$ except for the one corresponding to the entry face (which is never identified with any other face during decompression). This 2-bit *fold code* distinguishes faces on which the folding operation is to be executed (fold faces) from other faces and, for each fold face, identifies one of its edges as the fold edge. Gluing operations are encoded as two integers identifying the two external triangles to be glued and a two bit *glue code* which specifies the 'twist' which has to be applied to one of them before the identification (Figure 4).

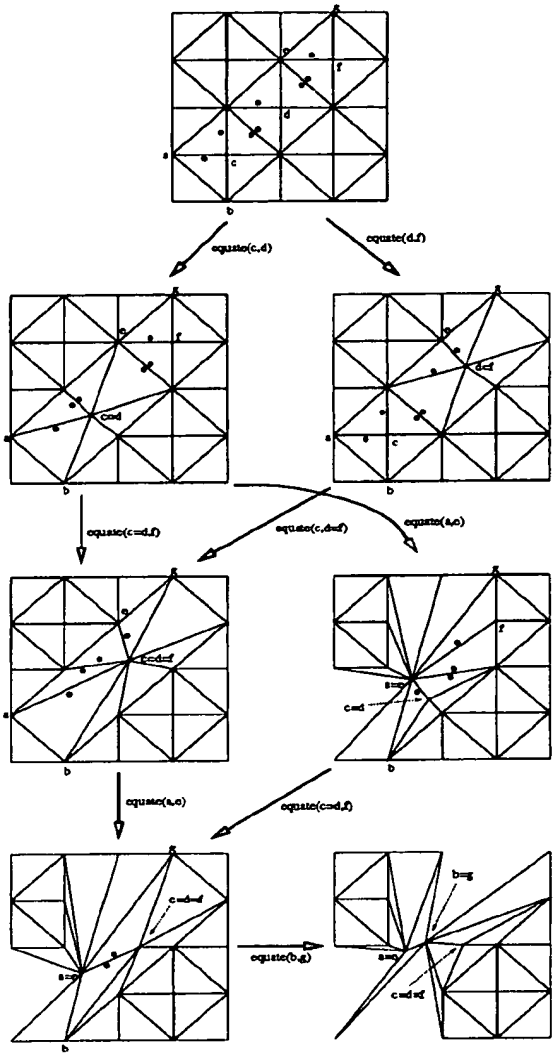

Figure 3: *Different ways of executing folding operations for a mesh whose boundary is shown on top lead to equating vertices in the same way (bottom right). Dots indicate the fold edge of a triangle and arrows - folding operations consistent with the folding scheme.*

Thus, gluing operations are considerably more expensive to encode. Fortunately, their number in a typical mesh is relatively small compared to the number of folding operations (for our test cases it was 200-700 times smaller), so that they usually do not contribute to more than 1-2% of the encoding size.

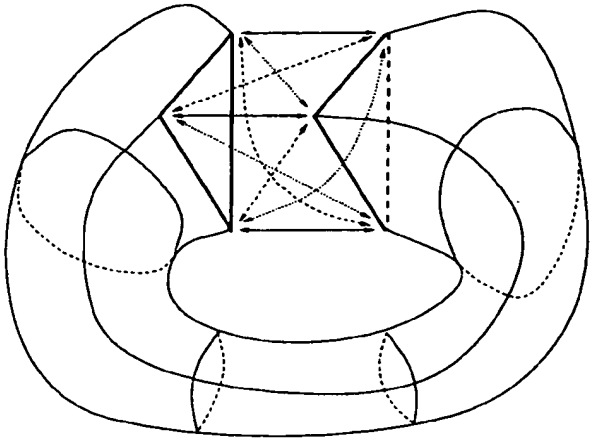

Figure 4: *Three possible ways of equating vertices of two glue triangles (shown by arrows of different styles). For each of the glue triangle pairs, one of them is specified by the glue code.*

4.3 Compression Results

The total size of our encoding is $7m + 2 + (\lceil \log_2(g + e - 1) \rceil + 1)g$, where m, g and e are the numbers of tetrahedra, glue faces of the tetrahedron tree and external faces of the original mesh. This cost can be broken as follows. The encoding of the tetrahedron spanning tree takes $3m$ bits. Storing the fold codes requires two bits per external face of the tetrahedron tree except for the one corresponding to the entry face. Since a tetrahedron has 4 external faces and attaching a tetrahedron to an external face increases the number of external faces by 2, the total number of external triangles in any tetrahedron tree with m tetrahedra is $2m + 2$. It follows that we need $2(2m + 1) = 4m + 2$ bits to store the fold codes. Fold triangles get nonzero fold codes, while all others (either glue, i.e. identified by means of gluing or corresponding to external faces of the original mesh) get the code of 00. Thus, we can specify a glue face using $\lceil \log_2(g + e - 1) \rceil$ bits, where by e and g we denote the number of external faces of the original mesh and glue faces of the tetrahedron tree (respectively). Including the two bit code specifying the twist, each glue triangle pair requires $2\lceil \log_2(g + e - 1) \rceil + 2$ bits to encode. The total size of the encoding of all glue triangle pairs is therefore $(\lceil \log_2(g + e - 1) \rceil + 1)g$ bits.

5 Details of our Approach

5.1 Compression

The compression procedure breaks into three major steps:

1. Building and encoding a tetrahedron spanning tree

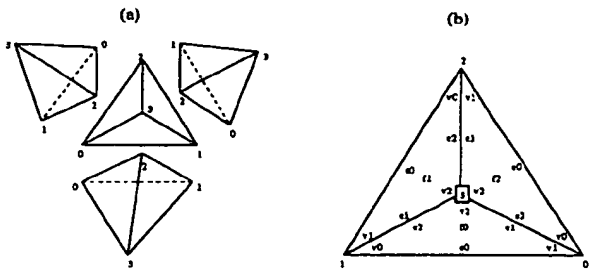

Figure 5: *(a) Induced orderings of vertices of the three branching tetrahedra (some of them may be missing from the actual tetrahedron tree); The door of the tetrahedron in the center (which is the parent of the surrounding three) is bounded by its three leading vertices (0,1, and 2). (b) Enumeration orders of three non-door faces (f0,f1,f2) and edges and vertices of each of these faces (e0,e1,e2 and v0,v1,v2), induced by the order of vertices of a tetrahedron (0,1,2,3)*

2. Creating a folding scheme

3. Building the folding string

The details of each of the above three steps are given below.

5.1.1 Building and Encoding a Tetrahedron Spanning Tree

In order to build a tetrahedron spanning tree one chooses an external triangle of the mesh to be the *entry face* and uses the incident tetrahedron as the root. Starting from the root, we traverse each tetrahedron once using a recursive procedure which systematically selects the next candidate from the undiscovered neighbors of the current tetrahedron. For a tetrahedron which is not the root, by its *door* we mean the triangle which separates it from its parent.

This recursive procedure corresponds to a depth-first search traversal of the dual graph of the mesh, in which nodes correspond to tetrahedra and links to triangles that separate two tetrahedra. Given a tetrahedron spanning tree, there are three types of triangle faces in the mesh.

- external faces (those on the boundary of the mesh),
- doors (triangles corresponding to tree edges of the dual graph of the mesh; equivalently those which are door faces to some tetrahedron),
- cut faces (all others, i.e. internal faces which are not doors).

The encoding of the tetrahedron tree is a sequence of triples of bits, one per tetrahedron, arranged in the traversal order. The i-th bit in a triple encodes whether

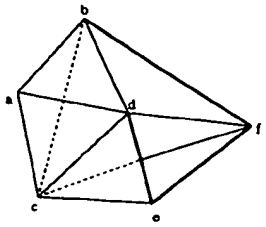

Figure 6: Free edges (bold lines) of the surface consisting of triangles abc,abd,acd,bcd,cde,cdf,cef,bdf

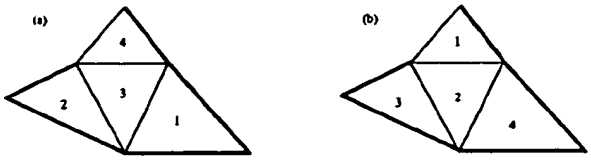

Figure 7: *The numbers of g-triangles for the two removal orders are (a) 2, (b) 0. The thick edges are external edges of the mesh. Our greedy strategy leads to the removal order shown in (b), possibly with the third and fourth triangle switched.*

the i-th non-door face of the corresponding tetrahedron is a door to some other tetrahedron. To make this precise, we need an enumeration order of non-door faces of each of the tetrahedra. In our implementation, this order is defined by the ordering of vertices of the tetrahedron, assigned to it as it is discovered during the traversal in the way shown on Figure 5. Let us note that our particular ordering is somewhat arbitrary and can be substituted by a different ordering convention.

We also use the traversal order of tetrahedra to obtain the order in which the vertices have to be rearranged before being made a part of the encoding string. Consider the sequence s of vertices obtained by concatenating sequences of vertices of all tetrahedra in the traversal order (for each single tetrahedron, we always list its vertices in the order assigned during the traversal). Clearly, the length of s is $4m$ and each vertex of the mesh appears as its entry. However, most vertices appear in it more than once (except for those which are vertices of precisely one tetrahedron). To get rid of the repeating vertices, we scan the sequence s, leaving only the entries encountered for the first time and removing all others. The resulting permutation of vertices defines the order in which the vertex data has to be transmitted to ensure correct reconstruction of the mesh geometry by the decompression algorithm.

5.1.2 Creating a Folding Scheme

Recall that the folding scheme imposes restrictions on the order of execution of folding operations so that the decompression procedure restores the structure of the original mesh from the tetrahedron tree. The process of building a folding scheme is essentially an inversion of the gluing and folding operations performed during decompression. This can be seen very clearly when one thinks of it in terms of the complex C resulting from the original mesh by cutting it along the *cut surface* formed by the cut triangles. To construct a folding scheme we delete the cut triangles one at a time. Such a removal of a cut triangle is equivalent to identifying the two external triangles of C which correspond to that triangle. If the identified triangles share an edge, the identification is a folding operation. This happens if and only if the removed triangle has a *free edge*, i.e. an edge which is an internal edge of the original mesh and, at the same time, is not shared with any other cut triangle (Figure 6). If this is the case, we mark t as an f-triangle (f for fold) and one of its free edges as its f-edge. Otherwise, t is classified as a g-triangle (g for glue). Clearly, the numbers of the f- and g-triangles depend on the order in which they are removed (Figure 7). Since glue triangles cost more and, as we shall see later, their number is twice the number of g-triangles, we would like to make the number of g-triangles as small as we can. In order to do that, we use a greedy strategy: we do not remove cut triangles with no free edges unless there is no other choice. For the surface on Figure 6, an example triangle removal order which may be produced by the above procedure is: bdf (bf becomes its f-edge), cdf (f-edge:df), ced (de), cef (any edge can be taken as its f-edge), abc (this is a g-triangle), abd (f-edge:ab), acd (ac or ad), bcd (any edge can be the f-edge).

5.1.3 Building the Folding String

First, we assign two-bit fold codes to all pairs (T, t) with T a tetrahedron and t any cut or external triangle (excluding the entry face) adjacent to it. In what follows, we shall call such pairs *cut pairs*. If t is either an external face or a g-triangle, the code is 00. If t is an f-triangle, the code depends on which of its edges is the f-edge. We assign the code of 01, 10 or 11 to the cut pair (T, t) according to whether the f-edge of t is its first, second or third edge (in the ordering of edges relative to T, cf Figure 5).

Apart from the fold code, we need to associate 2 extra bits of information with a g-triangle t (i.e. the glue code mentioned in Section 4.2). Since it is an internal triangle, there are exactly two tetrahedra T_1 and T_2 adjacent to it. Assuming that T_1 comes before T_2 in the traversal order, the two bit glue code simply encodes whether the first vertex of t in the order relative to T_1 matches its first, second or third vertex in the order relative to T_2.

The traversal order together with the orderings of ver-

tices of tetrahedra induce an ordering of cut pairs in the following way. If a tetrahedron T is traversed before T', then any cut pair whose tetrahedron is T precedes any cut pair whose tetrahedron is T'. For cut pairs with the same tetrahedra, i.e. of the form (T,t) and (T,t') we use the ordering of non-door faces of T to break the tie: (T,t) comes before (T,t') if and only if t precedes t' in the ordering of non-door faces T.

To obtain the folding string, we concatenate the fold codes of all cut pairs in the above order (obtaining a string of $4m+2$ bits) and the encodings of all g-triangles. The encoding of a g-triangle t consists of:

- The encoding of the two cut pairs having t as their triangle. This requires $2\lceil \log_2(g+e-1) \rceil$ bits, where e is the number of external faces of the original mesh and g is twice the number of g-triangles (equivalently, the number of glue triangles of the tetrahedron spanning tree reconstructed during decompression). This is because we encode each cut pair as an integer, being the number of cut pairs with the fold code of 00 preceding it in our order and there are $e+g-1$ such cut pairs.

- the two-bit glue code.

The resulting folding string takes $2(2m+1)+(\lceil \log_2(g+e-1) \rceil + 1)g$ bits.

5.2 Decompression

In order to restore the original mesh from its encoding we need to do the following:

1. Grow a tetrahedron tree based on the tetrahedron tree encoding

2. Read and interpret the folding string: classify the external triangles as glue, fold or boundary, assign fold edges to fold triangles, pair up glue triangles and assign a glue code to each pair

3. Initialize data structures representing the boundary of the mesh and keeping track of how vertices are equated

4. Glue, applying the correct twist determined by the glue code

5. Fold

6. Map the $m+3$ vertex labels in the tetrahedron tree table into n vertex labels corresponding to vertices of the decoded mesh

```
procedure grow_tree ( s: bit sequence )
                          : tetrahedron_table;
var
    t : tetrahedron_table;
    next_unused_reference,current_bit, i : integer;
    v0,v1,v2,v3 : integer; # vertex references
begin
    empty the stack and the table t;
    push(0,1,2);
    next_unused_reference := 3;
    current_bit := 0;
    while current_bit < length(s) do :
        (v0,v1,v2) := pop();
        v3 := next_unused_reference++;
        put (v0,v1,v2,v3) at the end of the table t;
        if s[current_bit+2]=1 then
            push(v2,v0,v3);
        if s[current_bit+1]=1 then
            push(v1,v2,v3);
        if s[current_bit]=1 then
            push(v0,v1,v3);
        current_bit += 3;
end;
```

Figure 8: *Growing a tetrahedron tree*

5.2.1 Growing a Tetrahedron Tree

The purpose of this part of the decompression algorithm is to build a tetrahedron tree based on the information provided by the tetrahedron tree string and to define the orderings of vertices and tetrahedra consistent with the orderings introduced during compression. The tree growing procedure (whose pseudocode is given in Figure 8) starts with a single tetrahedron and builds a tetrahedron tree by incrementally applying the attaching operation to it. In our implementation, the vertices are represented by integers which can be thought of as vertex labels, which are increasing integers for consecutive vertices as they are first encountered in their construction. A stack is used to keep triples of vertices defining attachable external triangles of the mesh. Initially, the mesh is empty and the stack contains a triple of vertices (represented by the integers 0,1,2) bounding the triangle corresponding to the entry face of the original mesh. The growing procedure pops a list of vertices from the stack and attaches a tetrahedron to the face bounded by those vertices. This is done by creating a new vertex (represented by the least nonnegative integer which has not been used to reference a vertex) and inserting it, together with the three popped vertices, into the tetrahedron table. Then, a triple of bits is read from the encoding string and, for each nonzero bit of that triple, a face of the newly added tetrahedron is pushed on the stack. The way in which the bits of the triple are as-

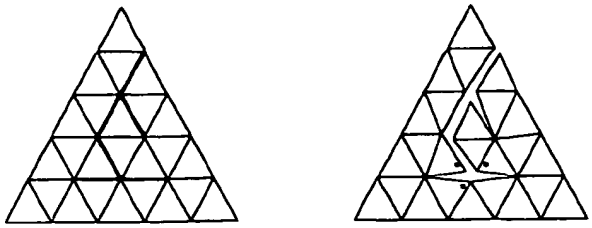

Figure 9: *A cut of a two-dimensional mesh along the bold edges; the three vertices of the complex on the right correspond to the same vertex of the original mesh.*

sociated with faces of tetrahedra as well as the order in which the vertices of the pushed triple are listed mimic the orderings used during compression. The output of the growing procedure is a tetrahedron table describing a tetrahedron tree together with an ordering of tetrahedra (given by the order in which they appear in the tetrahedron table) and ordering of vertices for each tetrahedron (the order in which they appear in a corresponding row of the table). Note that the decoding procedure is able to detect where the tetrahedron tree string ends without the need of any separator between it and the folding string.

5.2.2 Reading and Interpreting the Folding String

Although the tetrahedron tree grown based on the tetrahedron tree string contains all tetrahedra and some of the adjacency relations of the original mesh, they do not have the same structure unless the mesh is a tetrahedron tree itself. Geometrically speaking, the tetrahedron tree can be thought of as a result of cutting the original mesh along the surface formed by the cut triangles. A two dimensional example of cutting is shown in Figure 9. Cutting may replicate vertices (in the Figure, the three vertices of the cut marked with '*' are replica of the same vertex of the original mesh). The purpose of folding and gluing is to equate these replicated vertices to a single vertex.

The folding string is used to categorize the external triangles of the tetrahedron tree as fold, glue and boundary, corresponding to f-triangles, g-triangles and external triangles of the original mesh. To each fold triangle, one of its edges is assigned as the fold edge. The glue triangles are paired up and aligned using the two-bit glue code.

This is done by visiting the external faces of the tetrahedron tree in the same order as during compression and using the fold codes in the folding string to identify the face type and the fold edge for fold faces. Faces whose fold code is 01, 10 or 11 become fold faces and have the first second and third edge assigned as the fold edge. Faces with the fold code of 00 become either boundary or glue. Let l be the number of such faces. In order to distinguish boundary triangles from glue triangles we read the g-triangle encodings, which start with the $(4m+3)$-th bit of the folding string and occupy $2\lceil \log_2 l \rceil + 2$ bits each. Their interpretation is as follows. The first and second $\lceil \log_2 l \rceil$ bits encode two triangles with the fold code of 00, each one of them as an integer being the number of triangles preceding it with that fold code. These two triangles become a pair of glue triangles and they obtain the last two bits of the g-triangle encoding as their associated glue code.

5.2.3 Initialization of the Data Structure Representing the Boundary of the Mesh

The basic building blocks of the representation of the boundary of the mesh are:

- The triangle record, keeping three vertex references (integers, the same as those in the tetrahedron table), three pointers to adjacent edges and a set of flags allowing to determine if the triangle is a fold triangle and, if so, which of its edges is the fold edge.

- The edge record, keeping two pointers to adjacent triangles.

The construction of the above data structure can be implemented as a part of the tree growing procedure: it is initialized so that it describes the boundary of a single tetrahedron on startup and then updated right after each attaching operation.

Both gluing and folding identify two external triangles of the mesh and therefore change the structure of its boundary. Thus, the data structure storing the boundary of the mesh has to be updated after each glue or fold operation. For a fold operation, an update may be produced by an edge swap followed by an edge collapse (cf [12], [11]).

5.2.4 Mapping Vertices

While updating the boundary of the mesh, we equate the corresponding vertices of the identified triangles. Equating vertices with labels i and j is equivalent to replacing each occurrence of j in the tetrahedron table by i and subtracting 1 from all labels greater than j. Our actual implementation performs the label changes as a postprocessing step. More precisely, when we glue and fold, we maintain a graph whose vertices are labels $0, 1, \ldots, m+2$ and edges join the equated pairs of labels. After all gluing and folding operations are performed, we compute the mapping of the original labels into target ones by computing and ordering the connected components of the outcoming graph.

```
procedure glue;
begin
   for all glue pairs do :
      Let t and u be pointers to the two triangles in the
         pair, `t precedes `u in the tetrahedron table
         and v0,v1,v2 and w0,w1,w2 the vertex
         references of these triangles (in order);
      # twist according to the glue code
      if the glue code is 00, (ww0,ww1,ww2):=(w0,w2,w1);
      if the glue code is 10, (ww0,ww1,ww2):=(w2,w1,w0);
      if the glue code is 01, (ww0,ww1,ww2):=(w1,w0,w2);
      update the boundary of the mesh;
      equate pairs of references (v0,ww0), (v1,ww1)
         and (v2,ww2);
end;
```

Figure 10: *Gluing procedure*

5.2.5 Gluing and Folding

We start with performing gluing operations. We go over all glue triangle pairs and identify the two triangles in each pair and their corresponding vertices, updating our representation of the boundary of the mesh. The glue code of each pair provides information about what twist to apply before identifying the two triangles. The pseudocode which performs all the necessary gluing is given on Figure 10.

After all the gluing operations are done, we start folding. Recall that folding along an edge is allowed if and only if that edge is the fold edge of both adjacent external triangles. To avoid scanning edges in search of admissible fold ones, we adopt the following strategy. After any folding operation we recursively attempt to fold along the two external edges of the internal triangle resulting from the folding. In order to do all the folding operations, we call this recursive procedure for all external edges of the mesh.

It can be shown (see [24]) that, at this point, all glue and fold boundary faces have disappeared from the boundary of the mesh (i.e. have been identified with other faces becoming internal triangles). In other words, all exterior triangles of the current mesh are in fact boundary.

6 Complexity

In this section we argue that the compression and decompression algorithms can be implemented so that they both run in $O(s)$ time (respectively), where s is the encoding size (note that $s = O(m\log m)$ and $s = \Omega(m)$).

6.1 Compression

Building a tetrahedron spanning tree requires linear time in the number of tetrahedra, since the dual graph of the mesh has m vertices and $O(m)$ edges. In order to create a folding scheme, we remove cut triangles one at a time, always removing one with a free edge whenever possible. To implement this process so that it runs in $O(m)$ time one can use a procedure which removes a specified triangle and calls itself recursively for triangles adjacent to those of its edges which become free as a result of that removal. This recursive procedure is first called for all cut triangles with a free edge. After this is done, there are no cut triangles with a free edge left. To get rid of all cut triangles, we simply keep calling the above procedure for an arbitrarily chosen remaining cut triangle (which is then removed and tagged as a g-triangle). Equipping each triangle with an active flag which is reset when the triangle is deleted and storing a count of adjacent active cut triangles for each edge enables to delete cut triangles and test whether an edge is free or not in constant time. Since there are $O(m)$ triangles, the folding scheme can be constructed in $O(m)$ time. Assuming that it takes unit time to write a bit into a string, the process of creating the folding string takes $O(s)$ time.

6.2 Decompression

It takes linear time to build a tetrahedron tree and the data structure representing the mesh boundary (constant time update is necessary for each attaching operation). Reading and interpretation of the folding string takes $O(s)$ time. Each gluing and folding operation can be done in constant time and there are $O(m)$ of them. Thus, folding and gluing takes linear time in m. Similarly, vertex mapping takes $O(m)$ time since it boils down to computing connected components of a graph with $O(m)$ vertices and edges.

7 Discussion

In this section we discuss future work which may lead to improving the compression ratio achieved by the Grow&Fold algorithm.

First of all, one could encode the glue codes in a more compact way. Instead of using two bits to represent one of the three possible glue codes, it is possible to use only $\lceil \log_2 3^{(g/2)} \rceil = \lceil \frac{g \log_2 3}{2} \rceil$ bits to encode all of them. If the number of glue triangles is large, this leads to savings of over 0.4 bits per glue triangle pair.

Perhaps a more promising idea is to look for improvements of the coding scheme for tetrahedron trees. A simple observation that there are exactly $m-1$ bits set to one in our encoding of a tetrahedron tree leads im-

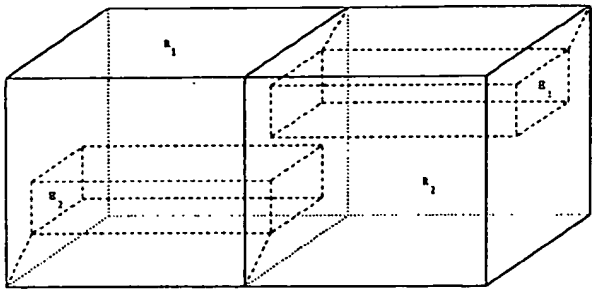

Figure 11: *The house with two rooms. It defines two 'rooms' R_1 and R_2. In order to enter R_i one has to walk through the corridor through the other room, starting with the 'door' E_i. The house consists of the walls of the two rooms with the corridors' entrances and exits removed, the corridors' boundaries and the two rectangular walls connecting each of the corridors to the outer wall.*

mediately to the conclusion that our encoding of tetrahedron trees is not optimal. Since ones appear in the tetrahedron tree string about twice as often as zeroes, entropy coding techniques allow to encode the tree with $3\log_2 3 - 2 \approx 2.75$ bits per tetrahedron. One may hope for even better results here, since not all sequences with exactly $m-1$ nonzero entries are a valid encodings of tetrahedron trees (any encoding string must have the property that there are at least k entries equal to one among the initial $3k$ symbols for any $k < m$).

Another interesting question concerns the number of glue triangles which are the reason for the nonlinear term in our estimate of the encoding length. Glue triangles are certainly needed for meshes with handles. Moreover, the number of glue triangles cannot be smaller than the number of handles. This is because each handle gives rise to a shell of cut triangles with boundary contained in the boundary of the mesh. No matter what the removal order of cut triangles during compression is, each such shell has to be broken at some point by removing a cut triangle with no free edges. However, even for meshes with no holes or handles glue triangles may be necessary. One can imagine a triangulation of the three dimensional ball and its tetrahedron spanning tree for which the cut triangles form a superset of the house with two rooms (cf [3, section I.2]): a two dimensional simplicial complex which does not cut the 3-space but whose triangulation has no triangle with a free edge (Figure 11). If this is the case, at least one glue triangle is needed (because the first triangle removed from the house must not have a free edge). However, it may be possible to change the tetrahedron tree so that no house with two rooms appears in the cut complex. For example, one could 'close' the entrance E_1 in Figure 11 and, simultaneously, remove some triangle from the wall of R_1. The resulting

n	m	l	l_g/l	l/m	T_c	T_d
100	514	3628	0.8%	7.058	0.09	0.04
1K	6298	44430	0.8%	7.055	1.19	0.52
10K	66487	469657	0.9%	7.064	14.99	6.27
50K	335188	2373774	1.2%	7.082	80.43	31.93
100K	672212	4767534	1.3%	7.092	166.84	70.84

Figure 12: *Experimental results*

space has the property that triangles can be removed from it one at a time in such a way that each removed triangle has a free edge. We see it as an indication that by changing the cut (or, equivalently, tetrahedron spanning tree) it is possible to decrease the number of glue triangles and therefore improve the performance of our coding algorithm. It would be interesting to develop an efficient algorithm which, by changing the tetrahedron tree, decreases the number of glue triangles (perhaps to the number which is optimal for the input mesh).

8 Experimental results

We tested our algorithm by running its prototype implementation for Delaunay tetrahedralizations of random sets of points in a cube. The tetrahedralizations were generated using the program qhull from the Geometry Center of the University of Minnesota. The results are given in Figure 12. In particular, they show that it is quite easy to obtain meshes which require nonzero number of g-triangles. However, the number of such triangles is relatively small, so that the encodings of the glue triangle pairs usually do not contribute to more than 1-2% of the total encoding size. The explanation of symbols used to describe the meaning of the columns of the table in Figure 12 is given below.

l - the total length of the encoding string

l_g - length of the encoding of the glue triangle pairs

n - number of points

m - number of tetrahedra

T_c - running time of the compression algorithm (in seconds)

T_d - running time of the decompression algorithm

The running times are the real time measurements. We ran our implementation on an SGI Power Challenge, with no significant effort to optimize or parallelize the code. One can notice that the running time growth is close to linear in the number of vertices of the mesh and that, for our test cases, the compression ratio never exceeded 7.1 bits per tetrahedron.

9 Conclusion

We discussed a simple topological compression scheme for connectivity of tetrahedral meshes which allows to store it using about 7 bits per tetrahedron. Our scheme can be compared to the standard representation via a tetrahedron-vertex incidence table, which requires $4\lceil \log n \rceil$ bits per tetrahedron, where n is the number of vertices of the mesh. We described efficient compression and decompression algorithms, running in linear time in the encoding size. We do not have a guaranteed linear bound on the encoding size, but the results of experiments with our prototype implementation show that our algorithm produces encodings whose length is nearly linear in the size of the mesh.

References

[1] B.G.Baumgart, Winged Edge Polyhedron Representation, AIM-79, Stanford University, Report STAN-CS-320, 1972.

[2] B.G.Baumgart, A Polyhedron Representation for Computer Vision, *AFIPS Nat. Conf. Proc.*, Vol.44, 589-596, 1975.

[3] M.M.Cohen, *A Course in Simple-Homotopy Theory*, Springer-Verlag 1970.

[4] M.Deering, Geometric Compression, *Computer Graphics (Proc. SIGGRAPH)*, p.13-20, August 1995.

[5] M.Denny and C.Sohler, Encoding a triangulation as a permutation of its point set, *Proc.9th Canadian Conference on Computational Geometry*, pp.39-43, Ontario, August 11-14, 1997.

[6] D.Dobkin and D.Kirkpatrick, A linear algorithm for determining the separation of convex polyhedra, Journal of Algorithms, Vol.6, pp.381-392, 1985.

[7] L.Floriani and B.Falcidieno, A Hierarchical Boundary Model for Solid Object Representation, ACM Transactions on Graphics 7(1), pp.42-60, 1988.

[8] E.Gursoz and F.Prinz, Boolean Set Operators on Non-Manifold Boundary Representation Objects, Computer-Aided Design 23(1), pp.33-39, January/February 1991.

[9] E.Gursoz, Y.Choi and F.Prinz, Node-Based Representation of Non-Manifold Surface Boundaries in Geometric Modeling, In: J.Turner, M.Wozny and K.Preiss eds., *Geometric Modeling for Product Engineering*, North-Holland 1989.

[10] P.Heckbert and M.Garland, Survey of Polygonal Surface Simplification Algorithms, in *Multiresolution Surface Modeling Course*, ACM SIGGRAPH Course Notes, 1997.

[11] H.Hoppe, Progressive Meshes, *Computer Graphics (Proc. SIGGRAPH)*, p.99-108, August 1996.

[12] H.Hoppe, T.DeRose, T.Duchamp, J.McDonald and W.Stuetzle, Mesh Optimization, *Computer Graphics (Proc. SIGGRAPH)*, p.19-26, August 1993.

[13] Y.E.Kalay, The Hybrid Edge: A Topological Data Structure for Vertically Integrated Geometric Modeling, Computer-Aided Design 21(3), pp.130-140, 1989.

[14] K.Keeler and J.Westbrook, Short Encodings of Planar Graphs and Maps, Discrete Applied Mathematics, No. 58, pp.239-252, 1995.

[15] D.Kirkpatrick, Optimal search in planar subdivisions, SIAM Journal of Computing, vol 12, pp 28-35, 1983.

[16] M.Mäntylä, *An Introduction to Solid Modeling*, Computer Science Press, Rockville, Maryland 1988.

[17] M.Naor, Succinct representation of general unlabeled graphs, Discrete Applied Mathematics, vol. 29, pp. 303-307, North Holland, 1990.

[18] R.Ronfard and J.Rossignac, Full-range approximation of triangulated polyhedra, *Proc. Eurographics'96, Computer Graphics Forum*, pp. C-67, vol.15, no.3, August 1996.

[19] J. Rossignac and M. O'Connor, SGC: A Dimension-independent Model for Pointsets with Internal Structures and Incomplete Boundaries, in Geometric Modeling for Product Engineering, Eds. M. Wosny, J. Turner, K. Preiss, North-Holland, pp. 145-180, 1989.

[20] J.Rossignac, Through the cracks of the solid modeling milestone, *From Object Modeling to Advanced Visual Communication*, Eds. S.Coquillart, W.Strasser, P.Stucki, Springer-Verlag, pp. 1-75, 1994.

[21] J.Rossignac, Edgebreaker: Compressing the connectivity of triangle meshes, GVU Technical Report GIT-GVU-98-17, Georgia Institute of Technology, `http://www.cc.gatech.edu/gvu/reports/1998`.

[22] J.Snoeyink and M.van Kerveld, Good orders for incremental (re)construction, Proc. ACM Symposium on Computational Geometry, pp.400-402, Nice, France, June 1997.

[23] O. Staadt and M. Gross, Progressive Tetrahedralization, Proc. IEEE Visualization, pp. 397:402, Research Triangle Park, October 18-23, 1998.

[24] A.Szymczak and J.Rossignac, Grow & Fold: Compression of Tetrahedral Meshes, GVU Technical Report GIT-GVU-98.

[25] G.Taubin and J.Rossignac, Geometric Compression Through Topological Surgery, ACM Transactions on Graphics, Vol.17, no.2, pp.84-115, April 1998.

[26] G.Taubin, W.Horn, F.Lazarus and J.Rossignac, Geometry Coding and VRML, Proceedings of the IEEE, pp.1228-1243, vol.96, no.6, June 1998.

[27] I. Trotts, B. Hamann, K. Joy, D. Wiley, Simplification of Tetrahedral Meshes, Proc. IEEE Visualization, pp. 287:295, Research Triangle Park, October 18-23, 1998.

[28] G.Turan, On the Succinct Representation of Graphs, Discrete Applied Mathematics 8, pp.289-294, 1984.

[29] T.C.Woo, A Combinatorial Analysis of Boundary Data Structure, IEEE Computer Graphics and Applications, Vol.5, pp.19-27, 1985.

Java Bytecode Compression for Low-End Embedded Systems

LARS RÆDER CLAUSEN, ULRIK PAGH SCHULTZ, CHARLES CONSEL, and GILLES MULLER
Compose Group, IRISA/INRIA

A program executing on a low-end embedded system, such as a smart-card, faces scarce memory resources and fixed execution time constraints. We demonstrate that factorization of common instruction sequences in Java bytecode allows the memory footprint to be reduced, on average, to 85% of its original size, with a minimal execution time penalty. While preserving Java compatibility, our solution requires only a few modifications which are straightforward to implement in any JVM used in a low-end embedded system.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Optimization*; *Intepreters*; *Run-time environments*

General Terms: Design, Experimentation

Additional Key Words and Phrases: Code compression, embedded systems, Java bytecode

1. INTRODUCTION

The Java language [Gosling et al. 1996], while enjoying widespread use in many application domains, is by design also meant to be used in embedded systems. This is witnessed by the availability of specific APIs, such as the JavaCard and EmbeddedJava specifications [Sun Microsystems, Inc. 1997; 1998a; 1999b; 1999c; 1999d]. The primary advantage of Java in this context is portability, which is realized through the Java bytecode format [Lindholm and Yellin 1996]. The use of a standard format allows any third-party developed services to be installed on any Java-compatible embedded system.

Low-end embedded systems, such as smart-cards, have strong restrictions on the amount of available memory, severely limiting the size of applications that they can run. Memory is scarce for a number of reasons: production costs must be kept low; power consumption must be minimized; and available physical space is limited. Thus, it is desirable that an embedded application consumes as little memory as possible, including the space taken up by the program code itself; the less space

This research was supported in part by Bull.
Authors' addresses: L. R. Clausen, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801; U. P. Schultz and G. Muller, IRISA/INRIA, Campus Universitaire de Beaulieu, F-35042 Rennes Cedex, France; C. Consel, LaBRI / ENSERB, 351 cours de la Libération, F-33405 Talence Cedex, France.

is taken by the code for each feature of the application, the more features can be embedded into the system.

Existing efforts at reducing the size of Java programs have concentrated on reducing the size of Java class files for transmission and subsequent execution on a standard workstation [Bradly et al. 1998; Pugh 1999]. In the Java class format, the constant pool comprises most of the space; the bytecode instructions only contribute about 18% of the total size [Antonioli and Pilz 1998]. However, the size of a class file is unimportant in the context of low-end embedded systems; only the memory footprint of the loaded program matters. In a low-end embedded system, the constant pool is either completely removed (when dynamic loading is not needed) or reduced using ad hoc techniques. We estimate[1] that the bytecode accounts for roughly 75% of the memory footprint in a system, using the token-based constant-pool approach of JavaCard 2.1 [Sun Microsystems, Inc. 1999d] (which allows dynamic loading of code).

Although Java bytecode is reasonably concise, programs still contain repeated patterns of code. Compression comes to mind as a viable solution for those situations where the size of the program code storage must be minimized. Data compression has a very wide range of applications, and is a well-studied area [Bell et al. 1990; Ziv and Lempel 1978]. A traditional solution would involve decompressing different parts of the program as they are needed, and discarding them afterward. However, this approach is usually not applicable in the context of low-end embedded systems. First, in a low-end embedded system, there may not be sufficient memory to decompress even a single method. For example, an existing JavaCard system such as the Java Ring is limited to 32K of ROM and 6K of RAM [Dallas Semiconductor Corp. 1998]. Second, the time taken to uncompress such a segment of code might exceed time constraints defined by the application domain.

This paper presents a solution that reconciles the need to conserve space on low-end embedded systems with fixed time constraints. We propose to factorize recurring instruction sequences into new instructions. This factorization allows more concise programs to run on a Java Virtual Machine (JVM) extended to support new instructions.

By expressing the new instructions as macros over existing instructions, the JVM only needs to be extended to support these macro instructions, not to support instructions specific to any one program. Using this technique, program memory footprint is on the average reduced to 85% of its original size, at an average run-time speed penalty between 2% and 30%.

The rest of the paper is organized as follows. Section 2 fixes the setting by discussing various applicable techniques for reducing the memory footprint of Java programs. We factorize code into macros over instructions, as illustrated in Section 3 by an example. Section 4 describes the actual factorization algorithm that we employ. Section 5 describes how macro support is implemented in the JVM. The experimental results are presented and discussed in Section 6. Finally, related work is described in Section 7, and concluding remarks are presented in Section 8.

[1]Based on measurements done using standard JavaCard CAP files, described in Section 6.

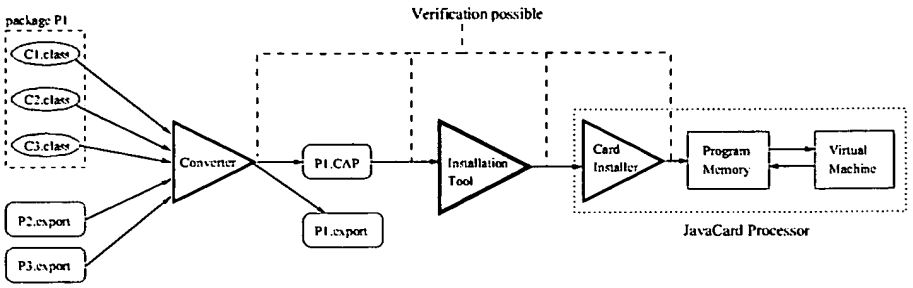

Fig. 1. Transferring Java classes into a JavaCard system.

2. APPROACHES TO REDUCING MEMORY FOOTPRINT

The standard Java class file format contains information that does not need to be present in a low-end embedded system. Thus, an internal, more compact format is used, as discussed in Section 2.1. However, the size of the bytecode can be further reduced, by using standard compression techniques in the limited fashion proposed in Section 2.2, or by factoring out recurring instruction sequences, as described in Section 2.3.

2.1 Conversion to an Internal Format

Although the Java bytecode instruction set was designed with embedded systems in mind, it is evident that standard Java class files produced by compilers such as Sun's `javac` compiler are not intended for use on such systems: debugging information and names of internal (private) identifiers are, for example, included by default. Although these are easily stripped from class files,[2] much precious space is still taken up by names that are not needed during execution. For this reason, it is natural for a low-end embedded system to use its own internal space-efficient representation. Throughout this paper, we will use the JavaCard 2.1 environment [Sun Microsystems, Inc. 1999b; 1999c; 1999d] as a reference, since it is the only documented, freely available low-end Java execution platform.

Java programs are transferred to JavaCard systems in units of packages, each package implementing either a set of applets or a library. The process of transferring a Java package to a JavaCard system is illustrated in Figure 1. First, a set of Java class files that make out a package is converted into a single CAP (converted applet) file and an export file describing the package interface. Export files describing other packages that are used by the classes in the package are also given to the converter. This scheme allows all the name information to be stored in export files, with two-byte tokens as the only representation of names in the CAP file. The CAP file is transferred onto the JavaCard device, which is then free to convert it into whatever internal representation is used for execution. Verification of the class files can be

[2]Numerous utilities, such as IBM's Jax (accessible from `http://www.alphaworks.ibm.com/`) reduce the size of Java class files by removing such superfluous information. Jax also performs class hierarchy specialization [Tip and Sweeney 1997], which removes unused features from Java programs, and is orthogonal to the techniques presented in this paper.

done at all stages in the process, but the properties that can be verified become more limited as more and more information is removed.

The low-level implementations of JavaCard systems are strictly proprietary, making it difficult to give a precise description of what internal format could be used to store Java programs. As an approximation, we use the standard CAP file format as in-memory format; information not needed after installation onto the card is assumed to have been stripped from the CAP file. Concretely, a CAP file is separated into several components, and we exclude those components not needed for execution when the CAP file format is the in-memory format. The stripped CAP file format is explained in the appendix, where a more detailed overview of the CAP file format also can be found. The stripped CAP file format is not ideal in terms of space consumption, and a realistic embedded system would probably use a more optimized format, giving a smaller memory footprint. Nonetheless, due to the lack of precise information regarding concrete embedded systems, the stripped CAP file format will serve as memory footprint measure throughout this paper. According to the experiments with stripped CAP files reported in Section 6, the bytecode takes up most of the memory footprint. We thus concentrate on reducing the size of the bytecode.

2.2 Basic-Block Compression

Looking beyond simple conversion of the Java class file into a more compact format, an often-used solution for compression is word-stream compression techniques such as Huffman encoding [Huffman 1952] or Lempel-Ziv compression [Ziv and Lempel 1978]. The bytecode of the whole program can be stored in compressed form, decompressing each part of the program from ROM to RAM, as it is needed during execution. However, given the limited memory resources of low-end embedded systems, it is not even possible to decompress each method as it is invoked. Rather than storing complete parts of the program in RAM, stream compression can be applied individually on each basic block of the code. Since the instructions in a basic block are used sequentially, they can be decompressed on-the-fly by a modified JVM, without having to store them to RAM. (The disadvantage is that decompression on-the-fly makes the overhead proportional to the program running time rather than the program size.) While such generic compression algorithms may not be optimal for the kinds of patterns found in program code [Ernst et al. 1997], they are well-known and can easily be implemented. However, because stream compression techniques are not well suited for the compression of many small, individual blocks, the expected gains in compression are limited.[3] Also, the restrictions on the amount of available RAM would impose strong restrictions on the size of the dynamic dictionary; these restrictions would have detrimental effects on the degree of compression. In addition, a significant time overhead would be associated with decompressing each basic block, slowing down the overall speed of the system to an unacceptable degree.

[3]On average Java methods are small, and basic blocks are even smaller. For example, in the programs used for experiments in Section 6, the average method length is roughly 50 bytes.

```
public class Point {
  public int x, y;
  public int dist() {
    return Z.intSqrt( (x*x)+(y*y) );
  }
}
```

```
public class Rectangle {
  public Point p1, p2;
}
```

Fig. 2. Java source code for the `Point` and `Box` classes.

2.3 Code Factorization

Most Java bytecode programs contain repeated occurrences of instructions. As a simple example, consider a specific object field that is manipulated throughout a class; furthermore assume that each access to this field is performed by the same sequence of operations. Common-subexpression elimination can be used to eliminate some of this redundancy. However, this optimization only applies to the rare cases where the instruction sequences actually compute the same value.

A simple way of eliminating code redundancy is to create methods that store repeated instruction sequences. Each original sequence of instructions is replaced by a call to such a method. However, there is a space overhead for defining a method and for invoking it (three bytes per invocation). Furthermore, any changes to the local state have to be copied explicitly back and forth, introducing significant time and space overheads. The code space overhead can be reduced to a few bytes per replaced instruction sequence by using Java bytecode subroutines instead of methods. However, such subroutines are intraprocedural, making the applicability of each subroutine too limited for our purposes.

As an alternative to a pure Java solution, our proposal is to extend the instruction set of the virtual machine with instructions that can replace recurring instruction sequences. In contrast to the workstation world, JVMs for embedded systems are proprietary and are as a rule written specifically for, and manually optimized to, each system. This makes it feasible to add new features to the JVM, as long as the changes are minimal and systematic, and as long as the JVM is still able to run standard Java bytecode.

Adding a fixed set of new instructions would be a nontrivial change that would significantly increase the size of the JVM. Furthermore, if the new instructions are specific to a given program, then they would have to be replaced if a different program is to be used. Our alternative is to extend the virtual machine to read new instruction definitions from the CAP file. These *macro instruction* definitions consist of bytecode instructions, and replace common instruction sequences in the code. Any instruction not in the standard instruction set is assumed to be a programmable instruction, defined by a table specific to the program being interpreted. The macro instructions can be stored in the run-time system, with very little memory overhead. With this approach, the number of new instructions is limited only by the number of instructions not used in the standard instruction set.

3. A SIMPLE EXAMPLE

As an example of our approach, we use the Java classes of Figure 2. The class `Point` represents a geometrical point, with a method that computes the distance to the

```
Method int dist()
  0 aload_0
  1 getfield #4 <Field Point.x I>
  4 aload_0
  5 getfield #4 <Field Point.x I>
  8 imul
  9 aload_0
 10 getfield #7 <Field Point.y I>
 13 aload_0
 14 getfield #7 <Field Point.y I>
 17 imul
 18 iadd
 19 invokestatic #6
    <Method Z.intSqrt(I)I>
 22 ireturn

Method Point()
  0 aload_0
  1 invokespecial #5
    <Method java.lang.Object.<init>()V>
  4 return

Method Rectangle()
  0 aload_0
  1 invokespecial #5
    <Method java.lang.Object.<init>()V>
  4 return
```

Fig. 3. Java bytecode for the `Point` and `Rectangle` classes.

center of the coordinate system. The class `Rectangle` represents a geometrical rectangle defined by two opposing corner points. The corresponding bytecode program is shown in Figure 3. Default constructors for both classes have been automatically introduced by the Java compiler.

In the bytecode program of Figure 3, there are some obvious opportunities for factorization. To access the `Point.x` field, the `Point` instance is loaded onto the stack, and a `getfield` instruction is used to extract the value. This field access yields two repetitions of the following instruction sequence:

```
0 aload_0
1 getfield #4 <Field Point.x I>
```

Furthermore, both classes have been extended with a default constructor, which consists of an invocation of the constructor of `Object`:

```
0 aload_0
1 invokespecial #5 <Method java.lang.Object.<init>()V>
4 return
```

Every default constructor in a program has exactly the same body, representing an ideal opportunity for factorization.

Faced with such sequences of generic instructions that are used repeatedly in specific programs, we replace each sequence by a new instruction. Let us now factorize the common instruction sequences identified in the program of Figure 3. Figure 4 shows the factorized bytecode program, along with the corresponding table of macros. The repeated instruction sequences for accessing fields have been factorized into macro instructions, as has the body of the constructors.

4. FACTORIZATION

We now present an algorithm for transforming a Java bytecode program into an equivalent program factorized with respect to a set of patterns. We give a high-level description of the algorithm used to obtain the results of this paper; implementation

```
Method int dist()
  0 Macro#204
  1 Macro#204
  2 imul
  3 Macro#205
  4 Macro#205
  5 imul
  6 iadd
  7 invokestatic #6
    <Method Z.intSqrt(I)I>
 10 ireturn
Method Point()
   0 Macro#206
Method Rectangle()
   0 Macro#206
```

```
Macro table:

Macro instruction 204:
   0 aload_0
   1 getfield #4 <Field Point.x I>
Macro instruction 205:
   0 aload_0
   1 getfield #7 <Field Point.y I>
Macro instruction 206:
   0 aload_0
   1 invokespecial #5
     <Method java.lang.Object.<init>()V>
   4 return
```

Fig. 4. Factorized Java bytecode for the `Point` and `Rectangle` classes.

details can be found in our technical report [Clausen et al. 1998]. Computing the optimal set of patterns is an NP-complete problem [Garey and Johnson 1979]. Our algorithm is designed to be simple and fast, while computing a set of patterns that is sufficient for the purpose of our experiments.

Conceptually, recurring sequences of operations are abstracted by factorizing them into single units. Each sequence of bytecode instructions is called a *pattern*. Factorizing a program with respect to a pattern yields a reduced program, where each *occurrence* of the pattern has been replaced by the corresponding new instruction. We refer to a control flow branch going from code surrounding an occurrence of a pattern into this occurrence as an *incoming branch*, whereas a branch going from a pattern to the code surrounding it is referred to as an *outgoing branch*. An outgoing branch is found in all occurrences of a pattern, whereas an incoming branch may be specific to a given occurrence of a pattern.

For a given program, factorization is done in two steps. First, repetitive instruction sequences are identified as patterns. Second, the bytecode is factorized with respect to these patterns, generating new instructions on-the-fly.

4.1 Pattern Generation

To find the set of patterns with which to factorize the program, all combinations of instruction sequences occurring in the program are generated; identical sequences are treated as a single *occurrence group*. First a group of length one is created for each set of equivalent instructions. These groups are iteratively expanded, either elongating each group or splitting it to create new groups of longer, equivalent occurrences.

Pattern generation must take into account how the constant pool is represented on the embedded system, to correctly reference constants after factorization. In the CAP file format, there is a separate constant pool for each package. The virtual machine keeps track of the current package to reference constants correctly, so two occurrences of an instruction in different packages with equal constant pool indices can be considered as being equal (permitting them to be factorized into the same pattern): the virtual machine will correctly interpret the instruction in the context of the current package. Alternative strategies to deal with constants in the virtual

machine and in the factorization algorithm are discussed in Section 5.3.

We reduce the occurrence groups to avoid unfactorizable instructions in the patterns, and to avoid branch instructions that cross the boundaries of an occurrence group. The instructions `tableswitch`, `lookupswitch`, `jsr`, and `ret` are considered unfactorizable: a switch instruction has alignment constraints that makes it difficult to place inside a macro, and a subroutine instruction causes problematic intraprocedural control flow. Neither of these instruction types are used very often, so we do not consider it worth the extra complexity to factorize them. Unfactorizable instructions are removed from a pattern by splitting the pattern into two new patterns, and splitting the occurrence group accordingly. Similarly, whenever an outgoing branch leaves a pattern, the branching instruction is removed from the pattern, creating two new patterns. Next, incoming branches are checked. Since incoming branches may be the result of a single branching statement, we only remove occurrences that have an incoming branch. The first instructions of an exception handler, and the first and last instructions of a code region where exceptions are caught, are treated in the same way as targets of incoming branches.

4.2 Pattern Application

Having computed the set of patterns, we now generate the macro instructions. Macros are generated greedily by selecting the occurrence group that gives most savings first and continuing until we either run out of unused instruction codes or occurrence groups that save space. We replace each occurrence by a macro instruction and update any other occurrences that contained the replaced occurrence to reflect this change.

The number of unused instructions in the instruction set determines the possible number of new macro instructions. The number of unused instructions depends on the Java platform used; it ranges from 52 to 152 free instructions (the various JavaCard instruction sets will be discussed in Section 6.1). Representing a macro as a single byte is simple and has very little overhead. However, to overcome the limit imposed by the number of free instructions, we may wish to define macros that have a two-byte instruction length. We refer to such macros as double-byte macros with a double-byte instruction coding (as opposed to single-byte macros with a single-byte coding). Although this coding yields less size reduction than using a single-byte coding, it is still worth doing in some cases. The loss in size reduction can be minimized by assigning the double-byte coding to macros that have few occurrences. The first byte of a double-byte macro is the instruction code, and the second byte indicates an offset into a secondary table of ordinary single-byte macros. This gives room for 255 more double-byte instructions for each free single-byte instruction code used this way.

Since macros by definition are nonrecursive, the factorization program also computes the maximal intraprocedural macro nesting. This information can be used to simplify the modifications that need to be made to the JVM. The factorized bytecode replaces the unfactorized bytecode in the method bytecode component, and the macro table is placed in a custom CAP file component (see the appendix). Alternatively, the unfactorized bytecode can also be kept, and a choice between what code to be used can be made during code installation, so a JVM without support for execution of factorized code still can run the program.

5. IMPLEMENTING AN EXTENSIBLE JVM

This section describes the changes that are needed to make a standard JVM extensible. A few simple changes must be made once in the interpreter main loop.

5.1 Macro Representation

We make a standard JVM extensible by enabling execution of macro instructions. A macro is essentially defined by two values: the instruction code and the body. The body of the macro is a code block which is terminated by the special instruction `macro_end`. It may contain other macro instructions. The set of macros is global to all packages.

Macros are stored using a standard file format, enabling the modified interpreter to use macros produced by any factorization program [Clausen et al. 1998]. When transferring a factorized package into an embedded system, the macros must be transferred as well. This transfer can be done automatically, since the factorized code and the macros are already present in the CAP file for the package.

Verification of the factorized bytecode before transfer into the embedded system must also take macros into account. A trivial preprocessor could expand the macros before verification is performed. As a result, existing bytecode verifiers could be used. Verification of factorized bytecode on the embedded system is more difficult, but the properties that are typically verified at this stage are usually fairly simple, making it possible for the verifier to directly process factorized code.

5.2 JVM Main-Loop Modifications

Basically, the JVM modifications consist of enabling the interpreter to detect and subsequently call macros, dispatching based on the instruction number. The macro call is always local to a method; to enable returning to the calling instruction, a small stack must save return addresses. Thus, each method invocation stack frame must contain a fixed-size macro call stack of program counters. Since macros are nonrecursive, the maximum stack depth can be computed by the factorization algorithm along with the set of macros, and be verified by the preprocessor for the verifier.

When a macro instruction is invoked, the current value of the program counter is pushed on the macro stack. Afterward, the program counter is set to point to the first instruction of the body of the executed macro. Execution continues in the macro until either the macro return instruction `macro_end` is executed, an exception is thrown, or a return is made from the current method.

When the `macro_end` instruction is executed, the program counter is reset to the top value of the program counter stack (which is popped), and execution continues at the next instruction. If a return is performed during the execution of a macro, control is transferred back to the caller, and the current stack frame is popped from the stack, disposing any program counters stored on the macro stack. Similarly, if an exception is thrown during the execution of a macro, (1) the entire stack of program counters is popped, (2) the program counter is reset to the last program counter popped from the stack, and (3) control is transferred directly to the appropriate exception handler.

Table I. Instruction Set Sizes for Java Platforms

Java platform	key	instructions used	instructions free
Standard Java (EmbeddedJava and up)	`java`	203	52
JavaCard 2.1, with integer support	`jc21+i`	184	71
JavaCard 2.1, without integer support	`jc21-i`	135	120
JavaCard 2.0	`jc20`	103	152

5.3 Constant Pool Representation

We assume that the virtual machine uses the CAP file format as its in-memory format, and therefore does not resolve constants before execution, which implies that the factorization algorithm does not need to resolve constants either. For this mechanism to work correctly, the virtual machine must keep track of the current package. Given that the virtual machine implicitly keeps track of the current class, and that the package of a class can be trivially found from the class itself, this requirement does not impose any significant overhead.

As an alternative to referencing constants indirectly through the constant pool, an embedded system can resolve all constants when a package is installed onto the card. Indeed, this appears to be the purpose of the CAP file `Reference location` component (see the appendix for details). If constants are resolved globally, then the virtual machine no longer needs to keep track of the current package to correctly reference constants. However, the factorization algorithm must be modified to resolve all constants before factorization, to ensure that the factorized code can be resolved correctly during installation onto the card. A macro must only be shared between two different packages if for both packages constants referenced by the macro resolve to the same global address. We believe that performing global constant resolution would have a positive impact on the number of recurring patterns in the code and thus on the compression ratio, since instruction sequences performing the same action would be identical. However, all experiments reported in this document are performed without constant resolution.

6. PERFORMANCE EVALUATION

We have implemented factorization of standard Java class files and extended the JVM of the Harissa environment [Muller and Schultz 1999] with macro support. The Harissa environment integrates an optimizing off-line Java compiler with an interpreter; the interpreter allows execution of dynamically loaded programs. Using these tools, we have performed a number of experiments to evaluate our factorization technique. In this section, we first present our considerations for what JavaCard instruction sets to include in our experiments; then we describe the actual experiments and their results; last we provide an assessment of the results.

6.1 The Various Java(Card) Instruction Sets

There are no less than four different instruction sets to consider when working with Java low-end embedded systems. Although only two of these are used in the latest specification of JavaCard (version 2.1), we consider it interesting to measure the effectiveness of our factorization algorithm on all of these instruction sets, since factorization could be used for non-JavaCard systems.

Table I shows the different Java instruction sets. The standard Java instruction set [Lindholm and Yellin 1996] (denoted by `java`) is used in large Java embedded systems (i.e., non-JavaCard systems) [Sun Microsystems, Inc. 1998a]. The JavaCard 2.0 instruction set (denoted by `jc20`) is a strict subset of the standard Java instruction set, where instructions for operations not supported by JavaCard systems have been removed [Sun Microsystems, Inc. 1997]. There are two versions of the JavaCard 2.1 instruction set, one with 32-bit integer support (denoted by `jc21+i`), and one without 32-bit integer support (denoted by `jc21-i`) [Sun Microsystems, Inc. 1999d]. Both JavaCard 2.1 instruction sets extend the `jc20` instruction set with a number of new instructions not found in the standard Java instruction set; these new instructions are intended to permit a more compact program representation. Many of the new instructions are parameterized and are thus more general than the macros generated by our approach; we investigate issues related to the the new JavaCard 2.1 instructions in Section 6.3.

6.2 Experiments

To test the effectiveness of the factorization algorithm and the execution speed of the resulting program, we consider the following program packages:

JavaCard 2.1 Library. The JavaCard 2.1 library classes, taken from the JavaCard 2.1 Development Kit [Sun Microsystems, Inc. 2000]. These libraries are normally placed on every JavaCard 2.1 system; they represent ideal candidates for factorization. The parts of the library that require 32-bit integer support (packages `impl` and `installer` from the `com.sun.javacard` package hierarchy) are excluded in the `jc21-i` experiments.

JavaCard Applets. The demonstration applets distributed with the JavaCard 2.1 Development Kit [Sun Microsystems, Inc. 2000].

JavaRing Applets. The sample JavaRing applets available from the iButton home page [Dallas Semiconductor Corp. 1999], updated for JavaCard 2.1 compatibility. These applets require integer support and are thus excluded from `jc21-i` experiments.

Plain JavaCard. JavaCard applets together with the JavaCard 2.1 library classes.

Full JavaCard. JavaCard applets and JavaRing applets together with the JavaCard 2.1 library classes.

JES. Sun's Java Embedded Server 1.0, a full-featured Web-server for embedded systems [Sun Microsystems, Inc. 1999a].

CaffeineMarks. Microbenchmark suite designed specifically for Java [Pendragon Software 1997] (the embedded version).

Javac. JavaSoft's JDK 1.0.2 Java compiler [Sun Microsystems, Inc. 1998c], packages `acm`, `java`, `javac`, and `tree` from the `sun.tools` package hierarchy.

Of these program packages, the JavaCard library, JavaCard applets, and JavaRing applets were only tested with the various JavaCard instruction sets, and tests including the JavaRing applets always use instruction sets with integer support. The last three tests (JES, CaffeineMarks, and Javac) were all done with the full Java instruction set.

Table II. Size in Bytes of Bytecode and Memory Footprint before and after Factorization (†: Estimated memory footprint, ‡: excluding certain classes; see text.)

		Unfactorized				Factorized					Compression	
	Instruction set	Memory footprint	Bytecode size	Average method size	Bytecode part of memory footprint	Macros defined	Bytecode size	Macro table size	Bytecode + macro table size	Memory footprint	Bytecode (after/before)	Memory footprint (after/before)
JavaCard Library	jc20	16071	11241	31	69.9%	186	8221	680	9276	14106	**82.5%**	**87.8%**
	jc21-i‡	4941	4390	16	88.8%	119	2944	400	3582	4133	**81.6%**	**83.6%**
	jc21+i	14283	9453	26	66.2%	96	7560	354	8109	12939	**85.8%**	**90.6%**
JavaCard Applets	jc20	5624	4067	75	72.3%	150	2325	586	3211	4768	**79.0%**	**84.8%**
	jc21-i	4886	3329	61	68.1%	119	1977	446	2661	4218	**79.9%**	**86.3%**
	jc21+i	4886	3329	61	68.1%	67	2260	329	2723	4280	**81.8%**	**87.6%**
JavaRing Applets	jc20	7314	6305	106	86.2%	150	3402	554	4256	5265	**67.5%**	**72.0%**
	jc21+i	6494	5485	92	84.5%	77	3335	363	3855	4864	**70.3%**	**74.9%**
Plain JavaCard	jc20	21695	15308	37	70.6%	202	11216	858	12481	18868	**81.5%**	**87.0%**
	jc21-i‡	9827	7719	24	78.5%	149	5253	607	6161	8269	**79.8%**	**84.1%**
	jc21+i	19169	12782	31	66.7%	83	10418	383	10970	17357	**85.8%**	**90.5%**
Full JavaCard	jc20	29009	21613	46	74.5%	225	15400	1027	16880	24276	**78.1%**	**83.7%**
	jc21+i	25663	18267	39	71.2%	102	14327	592	15126	22522	**82.8%**	**87.8%**
JES†	java	206143	153133	53	74.3%	102	115374	1667	117248	170258	**76.6%**	**82.6%**
Caf.Mark†	java	4067	3021	37	74.3%	68	1882	439	2460	3506	**81.4%**	**86.2%**
Javac†	java	121189	90025	79	74.3%	127	72022	606	72885	104049	**81.0%**	**85.9%**
Averages				50.9	74.3%						79.7%	84.7%

Table II shows the size of the bytecode in bytes and the total memory footprint, before and after factorization. The size of the bytecode is shown with and without the macro definition code included. The average method size is shown for reference, as is the percentage of memory footprint taken by the bytecode. The maximal macro stack nesting did not exceed four on any of these tests, and the factorization for all the tests reported in Table II was performed in less than 10 minutes on a 233MHz Pentium II machine. The memory footprint is given for stripped CAP files (as explained in the appendix). We cannot give a precise figure for the memory footprint of the three non-JavaCard programs, since they cannot be converted to CAP files (they all use non-JavaCard functionality). We observe that, on average, across all instruction sets, the bytecode accounts for 74.3% of the memory footprint. This figure is used as an estimate when reporting the memory footprint for these three test programs. As mentioned earlier, the factorization of programs in `jc21-i` instruction set is sometimes done on a more limited set of classes (JavaCard Library), or not included at all (JavaRing Applets and Full JavaCard).

We expect that the higher the number of unused instructions in an instruction set, the better the compression ratio. It can be seen that consistently across all of the JavaCard experiments, the `jc20` and `jc21-i` instruction sets give rise to better compression ratios than the `jc21+i` instruction set, due to the higher number of unused instructions. There is, however, no consistent difference between the compression obtained for the `jc20` and `jc21-i` instruction sets, which we attribute to the small difference in unused instructions. For the `jc20` and `jc21-i` instruction sets, the footprint is reduced on average to roughly 84% of its original size, whereas the reduction is closer to 86% of the original size for the `jc21+i` instruction set. The

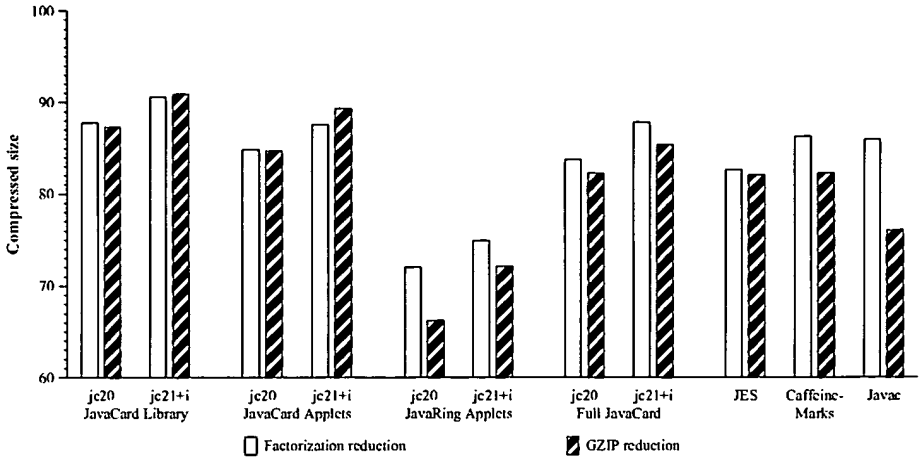

Fig. 5. Decrease in memory footprint by factorization and compression.

Table III. Factorization Options (factorization of some JavaCard 2.0 programs, memory footprint estimated, bytecode size includes macro table)

	bytecode size		memory footprint	
	bytes	effect	bytes	effect
unfactorized size	10754	—	15071	—
all options on (no CSE)	7733	0.0%	12050	0.0%
2-byte macros, nesting (no branches, no CSE))	7881	+1.9%	12198	+1.2%
2-byte macros, branches (no nesting, no CSE)	7928	+2.4%	12245	+1.6%
2-byte macros (no nesting, no branches, no CSE)	8006	+3.4%	12323	+2.2%
nesting, branches (no 2-byte macros, no CSE)	7826	+1.2%	12143	+0.8%
unfactorized size, CSE (vs. "unfactorized size")	10721	−0.3%	15038	−0.2%
all options on, CSE (vs. "all options on")	7748	+0.2%	12065	+0.1%

compression ratios for the Java programs are relatively high compared to the JavaCard programs, when taking into account the lower number of unused instructions. These programs were written with larger systems in mind, so it seems likely that the source code may contain more redundancy giving rise to more opportunities for factorization.

Factorization vs. `gzip` *Comparison.* To compare the compression obtained using factorization with an estimate of what would be gained by compressing each method individually using a standard compression algorithm, we compare the size reduction obtained with the standard Unix compression tool `gzip`. To use a `gzip`-like algorithm to uncompress methods in a JVM, each basic block would have to have been compressed individually using a global static dictionary, as was described in Section 2.2. To loosely estimate the size reduction obtained using this technique, we compress the bytecode of each method separately and subtract the overhead from the 20-byte header and file name. The compression ratios are shown in Figure 5. Factorization compression is roughly comparable to that of `gzip`, although `gzip` in general performs slightly better than factorization.

Assessment of Factorization Algorithm Features. The factorization algorithm factorizes out code containing branches, allows macros to be defined in terms of

Table IV. Benchmarks Comparing Normal Execution with Macro Execution

	Javac		CaffeineMarks (cm)	
	Pentium	SPARC	Pentium	SPARC
Without macros (n_1)	46.3s	69.6s	26cm	15cm
With macros (n_2)	46.6s	71.4s	21cm	11cm
Slowdown ($1 - n_1/n_2$)	2%	3%	19%	27%

other macros, and permits the use of two bytes for defining a macro instruction code. However, it is not obvious how much additional compression is provided by these features. Also, some standard optimizations such as common-subexpression elimination (CSE) tend to reduce code size, which might be cumulative with factorization. We test the advantage of each of the factorization algorithm features and the result of applying CSE in terms of additional compression, as illustrated in Table III. All experiments are performed in the `jc20` instruction set, and we perform CSE using the Cream bytecode optimizer [Clausen 1997]. Due to limitations of the Cream implementation, we cannot use the "Full JavaCard 2.1" benchmark. As an alternative, we use the JavaCard 2.0 library (taken from the JavaCard 2.0 Reference Implementation [Sun Microsystems, Inc. 1998b]) together with the JavaCard applets from our previous tests and the Visa Open Platform Card Version 1.0 implementation for Applet Developers [Visa International Service Association 1998] (a JavaCard 2.0-only library). Macros containing other macros offer the greatest advantage, followed by macros containing branch instructions, and finally double-byte macros. As for CSE, it reduces the unfactorized program size, but apparently conflicts with the factorization algorithm, and causes an increase in the size of the factorized program.

Run-Time Overhead of Factorized Code. To estimate the run-time speed overhead of using macros, we have performed two tests with the modified Harissa interpreter, both on Pentium (100MHz Dell Pentium) and SPARC (SPARC Station 5) architectures. Due to limitations in the Harissa interpreter, we were unable to run the Java Embedded Server. The first test measures the performance of the factorized Javac compiler. Although this is not a program that is likely to be placed in an embedded system, it is a large and complex application that performs a wide range of different data manipulation tasks. The result is shown in Table IV. There is virtually no slowdown, as compilation takes 2–3% longer when using the factorized code. The second test is the CaffeineMark benchmark. Given that the tests in this suite are microbenchmarks (i.e., tight loops testing very specific instructions), we assume that they represent a worst-case scenario with respect to the speed of factorized code. Here, we observe a slowdown of 19% on the Pentium and 27% on the SPARC. Code locality is strongly affected by factorization, which might have had a negative effect on our benchmarks. However, code locality is only an issue on systems with a cache, and most low-end embedded systems have a flat memory model.

6.3 Assessment

We have chosen the approach of generating bytecode macros over that of adding fixed instructions to the JVM. With the JavaCard 2.1 specification, the choice was

made to add new, fixed instructions to the JVM, offering a significant advantage in terms of reduced code size beyond what can be achieved using factorization, as was illustrated by our experiments. Naturally, factorization can still be applied to the extended instruction set, further reducing code size.

The additional compression offered by the JavaCard 2.1 instruction sets indicates that it would be worthwhile to allow parameterized macros. Parameterization could be in the form of a fixed number of instructions (possibly themselves macros), or in the form of arguments to bytecode instructions. Parameterization in the form of bytecode instructions should allow even more sharing of macro definitions, further improving compression. Parameterization in the form of bytecode instruction arguments would permit most JavaCard 2.1 instructions to be expressed in terms of a macro over JavaCard 2.0 instructions, allowing factorization to give compression comparable to that offered by the combination of factorization and the JavaCard 2.1 instruction set. Macros have a significant advantage over additional fixed instructions: a macro-enabled JVM is simpler to implement and takes up less ROM space than a JVM with a much larger instruction set (such as the JavaCard 2.1 instruction set).

7. RELATED WORK

The idea of compressing code is by no means new. Fraser, Myers, and Vendt describe an approach similar to ours, using suffix trees to compress assembly code [Fraser et al. 1984]. They get an average compression factor of 7%. They factorize local branches, use parameterized patterns, and implement a cross-jumping technique to exploit merging code sequences. Lefurgy, Bird, Chen, and Mudge replace common sequences of instructions with a single instruction macro [Lefurgy et al. 1997]. Compression is done on the instructions sets of the PowerPC, ARM, and i386 processors. However, minor hardware modifications are required for the compressed code to execute, contrary to the case for Java bytecode where only the virtual machine needs to be modified. The average compression rates obtained are 39%, 34%, and 26%, respectively.

Ernst et al. describe compression of code, both for transmission and for execution [Ernst et al. 1997]. They obtain the same compression ratio as `gzip` for executable code. They introduce a specific bytecode language for this purpose, using a bottom-up joining technique to form patterns. While their compression technique yields better results than ours, it requires greater amounts of RAM than is available on most low-end embedded systems.

Proebsting describes a C interpreter using "superoperators" [Proebsting 1995]. These kinds of operators can be automatically inferred from the tree-like intermediate representation produced by `lcc` [Fraser and Hanson 1991a; 1991b]. The operators are then used to produce an interpreter with specialized instructions. This transformation is aimed at improving speed; it gives a modest reduction in program size, at the cost of increased size of the generated interpreter. The approach of using a specialized interpreter could also be viable for embedded systems, e.g., by specializing the interpreter with respect to the general run-time environment.

As an alternative to the standard ZIP-based JAR format, Bradley, Horspool, and Vitek present the JAZZ format, which compresses collections of classes, improving the compression ratio by reorganizing data, so that similar data are compressed

together using standard compression techniques [Bradly et al. 1998]. Compression is about 75%, as opposed to only 50% for the standard JAR format. Pugh goes beyond this result by employing more aggressive compression techniques, completely reorganizing the class file layout, and employing dedicated compression techniques to each kind of data [Pugh 1999]. The resulting programs are on the average half the size of JAZZ-compressed programs. Although achieving superior compression ratios, neither of these techniques are appropriate for low-end embedded systems, both requiring more space and computation during uncompression than is available. Rayside, Mamas, and Hons present an alternative class file format that targets embedded systems that use the standard Java class file format as in-memory format [Rayside et al. 1999]. They primarily focus on reducing the size of the constant pool, while keeping it in a directly accessible format; class files are on the average reduced to 75% of their original size. While the class pool dominates the size of a Java program in the standard class file format, this is not true for low-end embedded systems. In effect, their technique is only relevant for embedded systems larger than those targeted by our factorization approach.

Franz and Kistler present SLIM binaries as an alternative to Java bytecode [Franz and Kistler 1997]. SLIM binaries provide a highly compact platform-independent structured program representation, designed to be translated into binary code by an optimizing just-in-time compiler. Due to the structured representation which can include information needed for optimizations, the compilation overhead is negligible, and the generated binary code is as efficient as that generated by an ordinary (optimizing) compiler. However, SLIM binaries are not easily interpretable in their compressed form, needing to be compiled into binary code before execution. This makes them unsuitable for low-end embedded systems.

Liao et al. optimize the selection of instructions on embedded DSP processors that have complex instruction sets defining compound operations [Liao et al. 1995]. Unlike the factorization process proposed in this paper where we generate new instructions for a given program, Liao et al. compile high-level programs to a given instruction set.

8. CONCLUSION AND FUTURE WORK

We have implemented factorization for Java bytecode. It handles nontrivial programs, and reduces the overall memory footprint of bytecode programs for low-end embedded systems to about 85% of their original size. Our factorization algorithm seems to compare satisfactorily with traditional compression, as embodied by `gzip`. The execution time overhead of introducing macros is between 2% and 30%. In particular, factorization can trivially provide better compression ratios than the pure-Java solutions proposed earlier (methods and subroutines) with a smaller run-time overhead, making it the embedded system engineer's preferred choice.

Inspired by the strong separation between packages in the JavaCard 2.1 specification, we are currently investigating the option of having package-local macro tables. This has several advantages, chiefly that dynamically loaded packages can be prefactorized using a private set of macros ranging over all free instructions. Also, we are investigating how to permit macros to take arguments, as described in Section 6.3. An initial assessment of the effect of having parameterized local vari-

able numbers for load and store instructions looks promising, but much work needs to be done to develop an efficient factorization algorithm. Finally, the factorization algorithm and the extensible JVM are by design independent of one another, allowing transparent replacement of our factorization algorithm with a new one that provides better compression. We consider the development of such factorization algorithms future work.

APPENDIX

To the authors' knowledge, all existing JVMs for low-end embedded systems are proprietary, making it difficult to assess the impact of bytecode compression on the memory footprint of a Java program. The JavaCard 2.1 CAP format is publicly specified [Sun Microsystems, Inc. 1999d], but it is not necessarily an appropriate format for the internal JVM data structures. It would be inaccurate to use a metric based on an existing JVM for a nonembedded environment (such JVMs usually are optimized for speed rather than space). For the lack of better concrete information we choose to use the CAP format as the basis for our memory footprint.

A CAP file consists of a number of *components* that in combination describe a complete JavaCard package [Sun Microsystems, Inc. 1999d]:

`Header`. General information about this CAP file and the package it defines.

`Directory`. Lists the size of each of the components defined in this CAP file, including any custom components.

`Applet`. Contains an entry for each of the applets defined in this package.

`Import`. Lists the set of packages imported by classes in this package.

`Constant pool`. Contains an entry for each of the classes, methods, and fields referenced by elements in the `Method` component of this CAP file.

`Class`. Describes each of the classes and interfaces defined in this package.

`Method`. The method component describes each of the methods declared in this package, including bytecode and exception handlers, but excluding `<clinit>` methods and interface method declarations.

`Static field`. Contains all the information required to create and initialize an image of all of the static fields defined in this package.

`Reference location`. Represents lists of offsets into the bytecode of the `Method` component to operands that contain indices into the constant pool array of the `Constant pool` component.

`Export`. Lists all static elements in this package that may be imported by classes in other packages.

`Descriptor`. Provides sufficient information to parse and verify all elements of the CAP file (optional component).

In addition, we use the CAP file custom component mechanism to store the macro table generated by the factorization algorithm in its own component. CAP files are actually generated in the Java JAR format, but will of course be decompressed during transfer to the JavaCard system. Thus, the memory footprint can be computed as the sum of the sizes of the decompressed CAP file components.

Not all of these components need to be included in the stripped CAP file format which we use to compute the memory footprint of a JavaCard system. The `Descriptor` component is only needed for verification, and is explicitly listed in the JavaCard 2.1 Virtual Machine Specification as being optional. The `Reference location` component has no obvious use except to perform global resolution of constants into absolute addresses. Since we lack more precise information, we assume that the CAP file format is the in-memory format used for execution. Hence, constants are not resolved into addresses, and the `Reference location` component is not needed. Thus, we exclude the `Descriptor` and `Reference location` components from our stripped CAP file format.

ACKNOWLEDGMENTS

We wish to thank Peter Chang for his proofreading assistance, and the anonymous TOPLAS referees for their helpful comments.

REFERENCES

ANTONIOLI, D. N. AND PILZ, M. 1998. Analysis of the java class file format. Tech. Rep. ifi-98.04, Department of Computer Science, University of Zurich. Apr. 28.

BELL, T. C., CLEARY, J. G., AND WITTEN, I. H. 1990. *Text Compression.* Prentice Hall.

BRADLY, Q., HORSPOOL, R. N., AND VITEK, J. 1998. Jazz: An efficient compressed format for Java archive files. In *Proceedings of CASCON'98.* Toronto, Ontario, 294–302.

CLAUSEN, L. R. 1997. A Java bytecode optimizer using side-effect analysis. *Concurrency: Practice & Experience 9,* 11 (Nov.), 1031–1045.

CLAUSEN, L. R., SCHULTZ, U., CONSEL, C., AND MULLER, G. 1998. Java bytecode compression for embedded systems. Research Report 3578, INRIA, Rennes, France. Dec.

Dallas Semiconductor Corp. 1998. *Java-Powered Decoder Ring.* Dallas Semiconductor Corp. URL: `http://www.ibutton.com/jring_facts.html`.

Dallas Semiconductor Corp. 1999. Java-Powered Ring Download Site. URL: `http://www.ibutton.com/jiblet`.

ERNST, J., EVANS, W., FRASER, C. W., LUCCO, S., AND PROEBSTING, T. A. 1997. Code compression. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Languages Design and Implementation.* ACM SIGPLAN Notices, vol. 32, 5. ACM Press, New York, 358–365.

FRANZ, M. AND KISTLER, T. 1997. Slim binaries. *CACM 40,* 12 (Dec.), 87–94.

FRASER, C. W. AND HANSON, D. R. 1991a. A code generation interface for ANSI C. *Software-Practice and Experience 21,* 9 (Sept.), 963–988.

FRASER, C. W. AND HANSON, D. R. 1991b. A retargetable compiler for ANSI C. *SIGPLAN Notices 26,* 10 (Oct.), 29–43.

FRASER, C. W., MYERS, E. W., AND WENDT, A. L. 1984. Analysing and compressing assembly code. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction.* ACM, 117–121.

GAREY, M. R. AND JOHNSON, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company.

GOSLING, J., JOY, B., AND STEELE JR., G. L. 1996. *The Java Language Specification.* The Java Series. Addison-Wesley, Reading, MA, USA.

HUFFMAN, D. A. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute for Radio Engineering 40,* 9 (Sept.), 1098–1101.

LEFURGY, C., BIRD, P., CHEN, I., AND MUDGE, T. 1997. Improving code density using compression techniques. In *Proceedings of the 30th Annual International Symposium on Microarchitecture.* IEEE Computer Society TC-MICRO and ACM SIGMICRO, Research Triangle Park, North Carolina, 194–203.

LIAO, S., DEVADAS, S., KEUTZER, K., AND TJIANG, S. 1995. Instruction selection using binate covering for code size optimization. In *Proceedings of the International Conference on Computer Aided Design*. IEEE Computer Society Press, Los Alamitos, Ca., USA, 393–401.

LINDHOLM, T. AND YELLIN, F. 1996. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA.

MULLER, G. AND SCHULTZ, U. 1999. Harissa: A hybrid approach to Java execution. *IEEE Software*, 44–51.

PENDRAGON SOFTWARE. 1997. CaffeineMark 3.0. URL: `http://www.webfayre.com/pendragon/cm3/index.html`.

PROEBSTING, T. A. 1995. Optimizing an ANSI C interpreter with superoperators. In *Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*. ACM Press, San Francisco, California, 322–332.

PUGH, W. 1999. Compressing Java class files. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI'99)*. Atlanta, Georgia, USA, 247–258.

RAYSIDE, D., MAMAS, E., AND HONS, E. 1999. Compact java binaries for embedded systems. In *Proceedings of CASCON'99*. Toronto, Ontario, 1–14.

Sun Microsystems, Inc. 1997. *JavaCard 2.0 Language Subset and Virtual Machine Specification*, 1.0 ed. Sun Microsystems, Inc. URL: `http://www.javasoft.com/products/javacard`.

Sun Microsystems, Inc. 1998a. *Embedded Java Specification*, 1.0 ed. Sun Microsystems, Inc. URL: `http://java.sun.com/products/embeddedjava/note.html`.

Sun Microsystems, Inc. 1998b. *Java Card API 2.0 Reference Implementation*, Developer Release 2 ed. Sun Microsystems, Inc. URL: `http://www.javasoft.com/products/javacard`.

Sun Microsystems, Inc. 1998c. *The Java Developers Kit 1.0.2*. Sun Microsystems, Inc. URL: `http://java.sun.com/products/jdk/1.0.2/`.

Sun Microsystems, Inc. 1999a. *Java Embedded Server*. Sun Microsystems, Inc. URL: `http://www.sun.com/software/embeddedserver`.

Sun Microsystems, Inc. 1999b. *JavaCard 2.1 Application Programming Interface Specification*. Sun Microsystems, Inc. URL: `http://www.javasoft.com/products/javacard`.

Sun Microsystems, Inc. 1999c. *JavaCard 2.1 Runtime Environment (JCRE) Specification*. Sun Microsystems, Inc. URL: `http://www.javasoft.com/products/javacard`.

Sun Microsystems, Inc. 1999d. *JavaCard 2.1 Virtual Machine Specification*. Sun Microsystems, Inc. URL: `http://www.javasoft.com/products/javacard`.

Sun Microsystems, Inc. 2000. JavaCard 2.1 development kit. URL: `http://www.javasoft.com/products/javacard`.

TIP, F. AND SWEENEY, P. F. 1997. Class hierarchy specialization. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-97)*. ACM SIGPLAN Notices, vol. 32, 10. ACM Press, New York, 271–285.

Visa International Service Association 1998. *Visa Open Platform Card Version 1.0 Implementation for Applet Developers*. Visa International Service Association. URL: `http://www.visa.com/cgi-bin/vee/nt/suppliers/open/main.html`.

ZIV, J. AND LEMPEL, A. 1978. Compression of individual sequences via variable rate encoding. *IEEE Transactions on Information Theory 24*, 530–536.

Received February 1999; revised September 2000; accepted March 2000